**1) Which type of OOP Concepts are used in your project?**

Your project utilizes multiple **Object-Oriented Programming (OOP) concepts**, including:

* **Encapsulation**:
  + You use **private fields** with **public getters and setters** in entities like Patient and Appointment, ensuring data is only modified through controlled methods.
* **Abstraction**:
  + Service classes (AppointmentService, PatientService) provide an **abstract layer** over the repository, hiding database complexities.
* **Inheritance**:
  + JpaRepository<Appointment, Long> and JpaRepository<Patient, Long> inherit methods from **Spring Data JPA**.
* **Polymorphism**:
  + **Method Overriding** is used where Spring Boot overrides save(), findById(), etc., from the JpaRepository interface.

**2) Which Type of Collection Do You Use in Your Project?**

Your project primarily uses **Java Collections Framework (JCF)**, particularly:

* **List (ArrayList/LinkedList)**
  + Example: List<Appointment> getAllAppointments() in AppointmentService
  + Used to store and return multiple Appointment and Patient objects.
* **Optional<T>**
  + Example: Optional<Patient> getPatientByEmail(String email)
  + Used for handling **nullable return values**, preventing NullPointerException.
* **HashMap<String, String>**
  + Example: In AuthController for storing authentication response.
  + Used to store key-value pairs (e.g., JWT token, email, role).

**3) Which Services Do You Provide in This Project?**

Your project provides **three main services**:

1. **Appointment Management (AppointmentService)**
   * Schedule appointments (saveAppointment).
   * Fetch all appointments (getAllAppointments).
   * Get appointments by email (getAppointmentsByPatient).
   * Delete appointments (deleteAppointment).
2. **Patient Management (PatientService)**
   * Register patients (savePatient).
   * Fetch patients (getAllPatients, getPatientById, getPatientByEmail).
   * Delete patients (deletePatient).
3. **Authentication (AuthController)**
   * Login using email and password (login).
   * Generate a dummy token ("dummy-jwt-token").

**4) Why Use JpaRepository Instead of CrudRepository?**

You use JpaRepository because it **extends** CrudRepository and provides **additional benefits**, such as:

✅ **Built-in Paging & Sorting (PagingAndSortingRepository)**

* JpaRepository includes **pagination and sorting** methods, unlike CrudRepository.

✅ **More Methods**

* JpaRepository provides:
  + findAll(Sort sort): Returns sorted results.
  + findAll(Pageable pageable): Returns paginated results.
  + flush(): Forces a flush of persistence context.
  + saveAndFlush(entity): Saves an entity and flushes changes instantly.

✅ **Efficient Batch Operations**

* Supports **batch deletes and updates** with deleteInBatch() and saveAll().

✅ **Better Performance for Large Data Sets**

* JpaRepository allows fine-grained control over **fetching strategies** (e.g., Lazy/Eager loading).

**In contrast, CrudRepository only provides basic CRUD (Create, Read, Update, Delete) operations, making JpaRepository more powerful and flexible.**

**4) Explanation of Annotations Used in Your Project**

Your project uses various **Spring Boot**, **Spring MVC**, **JPA**, and **Lombok** annotations. Here’s a detailed explanation:

**Spring Boot & Spring MVC Annotations**

1. **@SpringBootApplication**
   * Used in ClinicSystemApplication.java.
   * Marks this class as the main entry point for a **Spring Boot application**.
   * Combines:
     + @Configuration (enables Java-based configuration)
     + @EnableAutoConfiguration (automatically configures dependencies)
     + @ComponentScan (scans for components, controllers, services)
2. **@RestController**
   * Used in AppointmentController, AuthController, and PatientController.
   * Marks a class as a **Spring MVC Controller** where each method automatically returns a JSON response.
3. **@RequestMapping("/api/...")**
   * Defines **base URL** for a controller. Example:

@RequestMapping("/api/appointments")

* + All endpoints inside this controller will have URLs starting with /api/appointments.

1. **@CrossOrigin(origins = "http://localhost:5173")**
   * Enables **CORS (Cross-Origin Resource Sharing)**, allowing frontend (React at port 5173) to access backend APIs.
2. **@Autowired**
   * Used for **dependency injection**.
   * Example: AppointmentService is injected into AppointmentController:

@Autowired

public AppointmentController(AppointmentService appointmentService) {

this.appointmentService = appointmentService;

}

* + Allows Spring to **automatically instantiate** the service.

1. **@GetMapping("/{id}")**
   * Maps HTTP **GET requests** to a method.
   * Example:

@GetMapping("/{id}")

Public ResponseEntity<Appointment> getAppointmentById(@PathVariable Long id) {

* + If the user sends a request like GET /api/appointments/1, it extracts {id} as 1.

1. **@PostMapping**
   * Maps HTTP **POST requests** to a method.
   * Example: @PostMapping("/login") handles login requests.
2. **@RequestBody**
   * Extracts **JSON body** from an HTTP request and converts it into a Java object.
   * Example:

public ResponseEntity<?> saveAppointment(@RequestBody AppointmentRequest request)

* + Converts JSON { "email": "abc@example.com", "problem": "Fever" } into an AppointmentRequest object.

1. **@PathVariable**
   * Extracts **dynamic values** from a URL.
   * Example:

@GetMapping("/{id}")

public ResponseEntity<Appointment> getAppointmentById(@PathVariable Long id) {

* + If a request comes as GET /api/appointments/5, {id} becomes 5.

1. **@DeleteMapping("/{id}")**

* Maps HTTP **DELETE requests** to a method.
* Used for deleting appointments and patients.

**Spring Data JPA Annotations**

1. **@Entity**

* Marks a class as a **JPA entity**, meaning it is mapped to a database table.

@Entity

public class Appointment { ... }

* Creates an appointments table in the database.

1. **@Table(name = "appointments")**

* Specifies the **table name** for an entity in the database.
* Example: @Table(name = "patients") means the class Patient is mapped to a patients table.

1. **@Id**

* Marks a field as the **primary key**.

1. **@GeneratedValue(strategy = GenerationType.IDENTITY)**

* Specifies that the primary key should **auto-increment** in the database.

1. **@ManyToOne**

* Defines a **many-to-one relationship**.
* Example: Each Appointment is linked to **one** Patient.

@ManyToOne

@JoinColumn(name = "patient\_id")

private Patient patient;

1. **@Column(unique = true, nullable = false)**

* Specifies database constraints:
  + unique = true: Ensures no duplicate emails exist.
  + nullable = false: Email **must** be provided.

1. **@JsonFormat(pattern = "yyyy-MM-dd")**

* Formats **date fields** when sending JSON responses.

1. **@Temporal(TemporalType.DATE)**

* Ensures only the **date (without time)** is stored in the database.

**Lombok Annotations (For Boilerplate Code Reduction)**

1. **@Getter and @Setter**

* Auto-generates getters and setters for all fields.

**5) Flow of Your Project**

Your project follows the **MVC (Model-View-Controller) architecture**:

**1️ User Actions (Frontend)**

* A **React frontend** sends API requests to the **Spring Boot backend** via http://localhost:8808/api/.
* Example:
  + A user logs in via POST /api/auth/login.
  + A patient books an appointment via POST /api/appointments.

**2️ Controller Layer (Handles API Requests)**

* **Controllers** handle HTTP requests and responses:
  + AppointmentController → Manages appointment booking and retrieval.
  + PatientController → Manages patient registration and deletion.
  + AuthController → Handles authentication.

**3️ Service Layer (Business Logic)**

* Services **process requests** before interacting with the database.
* Example:
  + AppointmentService.saveAppointment() ensures data is valid before saving.

**4️ Repository Layer (Database Interaction)**

* **Repositories use JPA** to interact with the MySQL database.
  + JpaRepository automatically provides methods like save(), findById(), etc.
  + Example:

Optional<Patient> findByEmail(String email);

Fetches a patient based on email.

**5️ Database Layer (MySQL)**

* Data is stored in MySQL with the following tables:
  + **patients**
  + **appointments**

**Project Flow Summary (Example: Booking an Appointment)**

1️ **User (Frontend) sends a POST /api/appointments request** with { "email": "abc@example.com", "problem": "Flu" }.  
2️. **AppointmentController receives the request** and calls AppointmentService.saveAppointment().  
3️ **AppointmentService checks if the patient exists** using PatientRepository.findByEmail().  
4️ **AppointmentRepository.save() inserts the new appointment** into the database.  
5️ **A success response (200 OK) is sent back to the frontend** with the created appointment.

**6) you use hibernate how acctually it works explain me flow?**

**Hibernate Flow in Your Clinic Management System**

Hibernate is an **ORM (Object-Relational Mapping)** framework that helps convert Java objects into database tables and vice versa. Here's how Hibernate works in your project:

**1. Flow of Hibernate in Your Project**

**Step 1: Entity Class Creation (Mapping Java Classes to Database Tables)**

* You define **Entity classes** (Patient, Appointment) and annotate them with @Entity, @Table, and other JPA annotations.
* Example: The Patient entity is mapped to the patients table.

@Entity

@Table(name = "patients")

public class Patient {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long patientid;

@Column(unique = true, nullable = false)

private String email;

private String name;

private String lastname;

private String phoneNo;

private String gender;

private String city;

private String password;

}

**Step 2: Configuration (Hibernate + MySQL)**

* Hibernate needs **database configurations**, which are defined in application.properties:

properties

# Database Configuration

spring.datasource.url=jdbc:mysql://localhost:3306/clinic\_management?useSSL=false&serverTimezone=UTC

spring.datasource.username=root

spring.datasource.password=root

# Hibernate Configuration

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL8Dialect

spring.jpa.hibernate.ddl-auto = update

spring.jpa.show-sql=true

* spring.jpa.hibernate.ddl-auto = update → Automatically updates the database schema based on entity classes.
* spring.jpa.show-sql=true → Prints SQL queries executed by Hibernate.

**Step 3: Repository Layer (Connecting to Database)**

* **Spring Data JPA** repositories (PatientRepository, AppointmentRepository) extend JpaRepository to provide CRUD operations.
* Example:

@Repository

public interface PatientRepository extends JpaRepository<Patient, Long> {

Optional<Patient> findByEmail(String email);

}

* The repository layer communicates with the database using **Hibernate**.

**Step 4: Service Layer (Business Logic)**

* The PatientService and AppointmentService use @Service and interact with the repository.
* Example:

@Service

public class PatientService {

private final PatientRepository patientRepository;

public PatientService(PatientRepository patientRepository) {

this.patientRepository = patientRepository; }

public Patient savePatient(Patient patient) {

return patientRepository.save(patient); // Hibernate saves object to DB

}

}

* savePatient() calls save() on the repository, which **transforms the Java object into an SQL query**.

**Step 5: Controller Layer (Handling Requests)**

* @RestController handles **API requests** from the frontend and calls the service layer.
* Example:

@RestController

@RequestMapping("/api/patients")

@CrossOrigin(origins = "http://localhost:5173")

public class PatientController {

private final PatientService patientService;

public PatientController(PatientService patientService) {

this.patientService = patientService;

}

@PostMapping("/signup")

public ResponseEntity<?> registerPatient(@RequestBody Patient patient) {

return ResponseEntity.ok(patientService.savePatient(patient));

}

}

* When a request comes from **React Frontend**, Spring Boot receives it, processes it, and sends the data to Hibernate, which then saves/retrieves data from MySQL.

**7. Hibernate Execution Flow (Step-by-Step)**

1. **User sends an API request** (POST /api/patients/signup with patient details).
2. **Spring Boot Controller (PatientController)** receives the request.
3. **Service Layer (PatientService)** handles the business logic.
4. **Repository Layer (PatientRepository)** interacts with Hibernate.
5. **Hibernate converts Java objects into SQL queries** and executes them.
6. **MySQL stores/retrieves the data**.
7. **Hibernate converts the SQL result back into Java objects**.
8. **Spring Boot sends the response back to the frontend**.

**9. Example of How Hibernate Translates Java to SQL**

**When you call:**

Patient patient = new Patient();

patient.setName("John Doe");

patient.setEmail("john@example.com");

patientRepository.save(patient);

**Hibernate generates and executes:**

INSERT INTO patients (name, email, password, city, ...)

VALUES ('John Doe', 'john@example.com', 'hashed\_password', 'New York', ...);

**Similarly, when retrieving a patient:**

Optional<Patient> patient = patientRepository.findByEmail("john@example.com");

**Hibernate runs:**

SELECT \* FROM patients WHERE email = 'john@example.com';

**Explanation of Each Step:**

1️ Frontend (React App) → Spring Boot API

The React app sends a request (POST /api/patients/signup to register a patient).

2️ Controller Layer (PatientController)

The @RestController handles the API request and forwards it to the Service layer.

3️ service Layer (PatientService)

Contains business logic and calls the Repository to interact with the database.

4️ Repository Layer (PatientRepository)

The repository extends JpaRepository and calls Hibernate methods (save(), findByEmail(), etc.).

5️ Hibernate (ORM Layer) → MySQL Database

Hibernate converts Java objects to SQL queries and interacts with MySQL.

sql

INSERT INTO patients (name, email, password) VALUES ('John Doe', 'john@example.com', 'hashed\_password');

6️ Database Response → Hibernate → Repository → Service → Controller → Frontend

MySQL returns data, and Hibernate converts it back to a Java object.

The controller sends the response back to the React frontend.

8) How many methods are there in JpaRepository

JpaRepository Methods in Spring Boot

The JpaRepository interface in Spring Data JPA extends PagingAndSortingRepository, which in turn extends CrudRepository. So, it provides all the methods from CrudRepository and PagingAndSortingRepository, plus additional JPA-specific methods.

1️ Methods from CrudRepository<T, ID> (Basic CRUD Operations)

| **Method** | **Description** |
| --- | --- |
| save(S entity) | Saves a given entity. |
| saveAll(Iterable<S> entities) | Saves multiple entities at once. |
| findById(ID id) | Retrieves an entity by its ID. |
| existsById(ID id) | Checks if an entity exists by ID. |
| findAll() | Retrieves all entities. |
| findAllById(Iterable<ID> ids) | Retrieves all entities by their IDs. |
| count() | Returns the total number of entities. |
| deleteById(ID id) | Deletes an entity by its ID. |
| delete(T entity) | Deletes a given entity. |
| deleteAllById(Iterable<ID> ids) | Deletes multiple entities by their IDs. |
| deleteAll(Iterable<? extends T> entities) | Deletes multiple entities. |
| deleteAll() | Deletes all entities. |

2️ Methods from PagingAndSortingRepository<T, ID> (Pagination & Sorting)

| **Method** | **Description** |
| --- | --- |
| findAll(Sort sort) | Retrieves all entities sorted by the given criteria. |
| findAll(Pageable pageable) | Retrieves a paginated list of entities. |

3️ Additional Methods from JpaRepository<T, ID> (Advanced JPA Features)

| **Method** | **Description** |
| --- | --- |
| flush() | Flushes all changes to the database. |
| saveAndFlush(S entity) | Saves an entity and immediately flushes changes to the database. |
| deleteInBatch(Iterable<T> entities) | Deletes multiple entities in a batch. |
| deleteAllInBatch() | Deletes all entities in a single batch query. |
| getOne(ID id) | Retrieves a reference to an entity by ID (lazy loading). |
| findAll(Sort sort) | Finds all entities sorted by the given criteria. |
| findAll(Pageable pageable) | Finds all entities with pagination and sorting. |

4️ Custom Query Methods (Derived Query Methods)

Apart from built-in methods, you can define custom methods in JpaRepository using method names based on field names:

| **Method Name** | **Description** |
| --- | --- |
| findByName(String name) | Finds entities where name matches. |
| findByEmail(String email) | Finds a Patient by email. |
| findByCity(String city) | Finds all entities where city matches. |
| findByNameContaining(String name) | Finds all entities where name contains a given string. |
| findByGenderAndCity(String gender, String city) | Finds entities based on gender and city. |

Total Number of Methods in JpaRepository

JpaRepository inherits methods from CrudRepository and PagingAndSortingRepository.

CrudRepository → 11 methods

PagingAndSortingRepository → 2 methods

JpaRepository adds 4 more methods

Total 17+ methods (excluding custom queries).

9) Custom Queries in JpaRepository Using @Query Annotation

**📌 Understanding @Query Annotation in Easy Words**

Spring Data JPA allows us to **write custom queries** using the @Query annotation. This is helpful when **method name queries (like findByCity()) are not enough**.

**1️ JPQL (Java Persistence Query Language) Queries**

JPQL is **like SQL**, but instead of using **table names**, it uses **entity class names**.

**🔹 Example: Find Patients by City**

@Repository

public interface PatientRepository extends JpaRepository<Patient, Long> {

@Query("SELECT p FROM Patient p WHERE p.city = :city")

List<Patient> findPatientsByCity(@Param("city") String city);

}

✅ **What’s happening here?**

* @Query("SELECT p FROM Patient p WHERE p.city = :city") → **Custom JPQL query** to find patients by city.
* :city → **Dynamic parameter** (value will be passed at runtime).
* @Param("city") → Maps the method parameter to the query.

📌 **Example usage in service class:**

List<Patient> patients = patientRepository.findPatientsByCity("New York");

**2️ Native SQL Queries**

If you want to write **direct SQL queries** (instead of JPQL), use nativeQuery = true.

**🔹 Example: Find Patient by Email**

@Query(value = "SELECT \* FROM patients WHERE email = :email", nativeQuery = true)

Patient findPatientByEmail(@Param("email") String email);

✅ **Difference from JPQL:**

* Uses **table name** (patients) instead of entity class (Patient).
* Uses nativeQuery = true to **tell Spring that this is raw SQL**.

📌 **When to use native queries?**

* When JPQL **is not enough** (e.g., performance optimizations).
* When working with **complex SQL queries**.

**3️ Queries with Multiple Conditions (AND, OR)**

You can **filter results** based on multiple conditions.

**🔹 Example: Find Patients by City & Gender**

@Query("SELECT p FROM Patient p WHERE p.city = :city AND p.gender = :gender")

List<Patient> findPatientsByCityAndGender(@Param("city") String city, @Param("gender") String gender);

✅ **Explanation:**

* Finds **only patients who match both conditions** (city & gender).

**🔹 Example: Find Appointments Within a Date Range**

@Query("SELECT a FROM Appointment a WHERE a.appointmentDate BETWEEN :startDate AND :endDate")

List<Appointment> findAppointmentsBetweenDates(@Param("startDate") LocalDateTime startDate,

@Param("endDate") LocalDateTime endDate);

✅ **How it works?**

* BETWEEN :startDate AND :endDate → Filters appointments **within the given date range**.
* This is useful for **getting appointments for a specific week or month**.

**4️ Using LIKE for Partial Matching**

If you want to **search for names that contain a certain word**, use LIKE.

**🔹 Example: Search Patients by Name**

@Query("SELECT p FROM Patient p WHERE p.name LIKE %:keyword%")

List<Patient> searchPatientsByName(@Param("keyword") String keyword);

✅ **How it works?**

* %:keyword% → Matches any text **before or after** the keyword.
* If keyword = "Joh", it will find:
  + **John**
  + **Johnny**
  + **Johnson**

📌 **Use case:** Useful for **search bars** where users type a partial name.

**5️ Pagination with @Query**

If you have **thousands of records**, fetching everything at once is **slow**.  
Instead, use **pagination** to get data in small chunks.

**🔹 Example: Paginate Patients by City**

import org.springframework.data.domain.Page;

import org.springframework.data.domain.Pageable;

@Query("SELECT p FROM Patient p WHERE p.city = :city")

Page<Patient> findPatientsByCity(@Param("city") String city, Pageable pageable);

✅ **Why use pagination?**

* **Loads data faster** (fetches only required records).
* **Improves performance** (especially with large datasets).

📌 **Example usage:**

Pageable pageable = PageRequest.of(0, 10); // Get first 10 patients

Page<Patient> patients = patientRepository.findPatientsByCity("New York", pageable);

* PageRequest.of(0, 10) → **Fetches first 10 records**.
* Page<Patient> → Returns **paginated results**.

**✅ When Should You Use @Query?**

| **Scenario** | **Use @Query?** |
| --- | --- |
| Simple findByField() queries | ❌ No (Use method name queries) |
| Complex conditions (AND, OR, BETWEEN) | ✅ Yes |
| Joins between multiple tables | ✅ Yes |
| Custom sorting or grouping | ✅ Yes |
| Native SQL queries (for performance tuning) | ✅ Yes |

**Summary**

* @Query is used when **method name queries are not enough**.
* **JPQL works with entity names**, while **native queries use table names**.
* LIKE helps **search for partial matches**.
* **Pagination** improves performance by fetching **limited records**.

✅ **Now you can write efficient custom queries in Spring Data JPA!**